# An Efficient Communication Strategy for Finite Element Methods on the Connection Machine CM-5 System

# An Efficient Communication Strategy for Finite Element Methods on the Connection Machine CM-5 System

Zdeněk Johan
Kapil K. Mathur
S. Lennart Johnsson
Thomas J.R. Hughes

TR-11-93

April 1993


Parallel Computing Research Group

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

# An Efficient Communication Strategy for Finite Element Methods on the Connection Machine CM-5 System

Zdeněk Johan, Kapil K. Mathur, S. Lennart Johnsson

*Thinking Machines Corporation*
*245 First Street, Cambridge, MA 02142, USA*

Thomas J.R. Hughes

*Division of Applied Mechanics, Stanford University*
*Durand Building, Stanford, CA 94305, USA*

September 8, 1994

*Thinking Machines Technical Report No. 256*

*Submitted to:*
*Computer Methods in Applied Mechanics and Engineering*

**Abstract.** The objective of this paper is to propose communication procedures suitable for unstructured finite element solvers implemented on distributed-memory parallel computers such as the Connection Machine CM-5 system. First, a data-parallel implementation of the recursive spectral bisection (RSB) algorithm proposed by Pothen *et al.* is presented. The RSB algorithm is associated with a node renumbering scheme which improves data locality of reference. Two-step gather and scatter operations taking advantage of this data locality are then designed. These communication primitives make use of the indirect addressing capability of the CM-5 vector units to achieve high gather and scatter bandwidths. The performance of the proposed communication strategy is illustrated on large-scale three-dimensional fluid dynamics problems.

## 1. Introduction

One strength of the finite element method is its ability to handle unstructured meshes. Unfortunately, the resulting increased complexity of data structures may appear as a major issue in implementing finite element software on massively parallel computers since global communication between processors may be required. In addition, the current compiler technology is still immature when it comes to generating optimal code for general communication patterns. A detailed analysis of finite element techniques shows the need for only a small set of communication primitives. These primitives can therefore be written and optimized for each hardware platform and can be made available to users through library calls. Communication libraries for unstructured mesh solvers have already been proposed by Mathur and Johnsson [19] for the Connection Machine CM-200 system and by Sussman *et al.* [25] for the Intel iPSC/860 and the Intel Delta machines. In this paper, we propose a set of communication routines for finite element solvers running on distributed-memory computers. Even though we tailored their implementation to the Connection Machine CM-5 system hardware, the ideas presented herein can be applied to any distributed-memory massively parallel system. As current and future (at least in the foreseeable future) massively parallel computers are expected to have substantially higher local memory bandwidths than sustained network bandwidths, we designed these communication primitives to take maximum advantage of the fast processor-to-memory bandwidths.

An outline of this paper follows: After a brief description of the CM-5 system hardware, we present in Section 3 a parallel implementation of the recursive spectral bisection algorithm. A heuristic node renumbering algorithm designed to improve data locality of reference is proposed in Section 4. A description of the actual communication routines handling the gather and scatter operations found in finite element software is given in Section 5. In Section 6, numerical examples illustrate the performance of the proposed strategy. Finally, conclusions are drawn in Section 7.

## 2. The Connection Machine CM-5 system

The Connection Machine CM-5 system [28] is a multiple-instruction multiple-data (MIMD) massively parallel computer. The largest system built to-date has 1,024 processing nodes. The architecture is theoretically scalable to 16,384 processing nodes. Each processing node is composed of a SPARC processor controlling four vector units, 32 Mbytes

of memory divided into four banks, and a network interface for communication between processing nodes. Each memory bank is connected to a vector unit by a 64-bit wide data path. The SPARC processor, the vector units and the network interface are interconnected by a 64-bit wide bus. Each vector unit has a peak computing performance of 32 Mflops/s and a 128 Mbytes/s local memory bandwidth.

A CM-5 system contains one or more control processors that manage the processing nodes and the input/output subsystem. The control processors and processing nodes are interconnected by scalable control and data networks having a 4-ary fat-tree structure [17]. The control network is used for global operations and synchronization of the processing nodes. The data network allows memory-to-memory data transfer. A diagnostic network invisible to the user is used by the system manager to monitor the well-being of the CM-5 processing nodes. A schematic of the CM-5 architecture is shown in Figure 1. The control processors run an extended version of UNIX, and each processing node runs an operating system microkernel. Furthermore, the CM-5 system possesses timesharing and batch capabilities comparable to those found on conventional computers. For conciseness, a Connection Machine CM-5 system having $p$ processing nodes will be called a $p$-node CM-5 system.

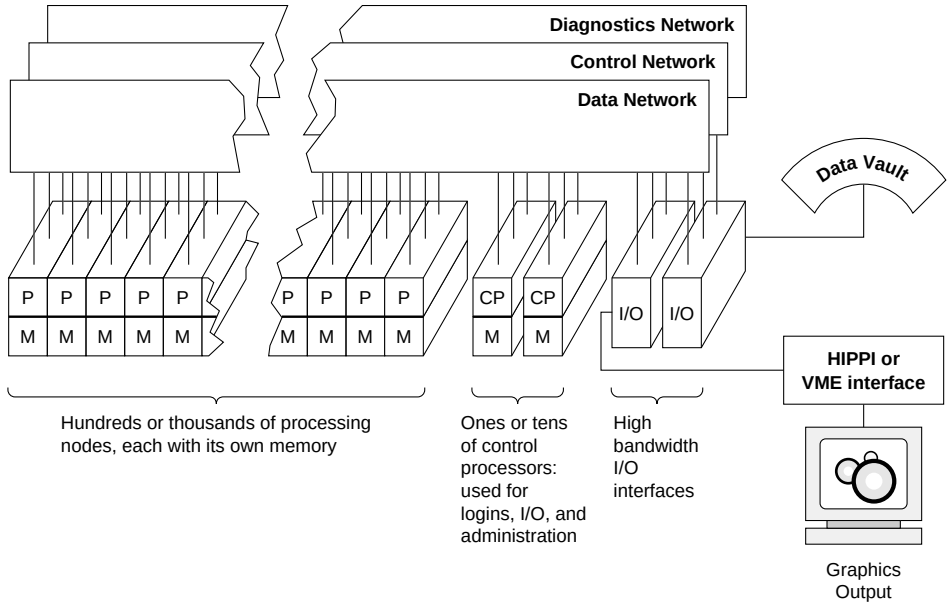


*Figure 1.* Schematic of the Connection Machine system CM-5 architecture.

We have chosen to use the Connection Machine Fortran language [30] (abbreviated

as CM Fortran or CMF) for most of our implementations. CM Fortran possesses the array syntax, the dynamic memory allocation capability, and most of the intrinsics defined in the Fortran 90 language. It also allows for a detailed control of data mapping onto the distributed-memory architecture through compiler directives.

## 3. Parallel recursive spectral bisection algorithm

The recursive spectral bisection (RSB) algorithm was proposed by Pothen *et al.* [21] as the basis for computing small vertex separators for sparse matrices. Simon [24] applied this algorithm to mesh decomposition and showed that spectral bisection compared favorably with other decomposition techniques. Venkatakrishnan *et al.* [27] and Das *et al.* [3] used the RSB algorithm in conjunction with unstructured finite volume Euler codes on the Intel iPSC/860. The major drawback of the RSB algorithm is its high computing cost, as noted in [24], caused by the need for solving a series of eigenvalue problems. It is often stated that a finite element mesh can be decomposed after it is generated, and the decomposition reused for the different calculations performed on that mesh. However, a new partitioning is to be obtained if adaptive mesh refinement is required. The mesh also has to be re-decomposed if the CM-5 configuration (i.e., the number of processing nodes available to the user) is changed between two calculations. In order to avoid the mesh decomposition from becoming a significant computational bottleneck, an efficient data-parallel implementation of the RSB algorithm using the CM Fortran language is developed. Improvements to the RSB algorithm possibly leading to better partitions and lower computing costs have been recently proposed by Hendrickson and Leland [9] and Barnard and Simon [1]. Unfortunately, some issues remain unresolved when it comes to implementing these new ideas on parallel computers. We therefore consider only the RSB algorithm in its simplest form for our implementation.

### 3.1. General issues

The first step is to define a good representation of the mesh element topology. This is done through the ***dual mesh connectivity*** array `idual` of dimension $n_{\text{faces}} \times n_{\text{el}}$ which contains the list of elements sharing a face with a given element. $n_{\text{faces}}$ is the number of element faces (e.g., $n_{\text{faces}} = 4$ for a tetrahedron and $n_{\text{faces}} = 6$ for a brick); and $n_{\text{el}}$ is the number of elements. An element having a face on the mesh boundary has its corresponding entry in `idual` set to zero. The purpose of the RSB algorithm is to generate a reordering $\boldsymbol{P}$ of the elements based on `idual` such that nicely shaped partitions of adjacent elements

are obtained (the quality of a partition will be defined in terms of communication cost in Section 5). These partitions are then mapped to the vector units of the CM-5 system, with the constraint of having at most one partition per vector unit. The number of partitions $n_{\mathrm{par}}$ and the number of elements per partition $\eta_{\mathrm{el}}$ are actually determined from the ***block distribution*** format used by the CM-5 run-time system. In this format, the $n_{\mathrm{par}} - 1$ first partitions contain the same number of elements, the last partition having the remaining elements. This leads to having $\eta_{\mathrm{el}} = \lceil n_{\mathrm{el}}/n_{\mathrm{vu}} \rceil$ elements in the first $n_{\mathrm{par}} - 1$ partitions, with $n_{\mathrm{par}} = \lceil n_{\mathrm{el}}/\eta_{\mathrm{el}} \rceil$. $n_{\mathrm{vu}}$ is the number of vector units in the CM-5 configuration considered. It should be noted that our parallel implementation of the RSB algorithm is tightly linked to the data mapping format described above. Major changes to the implementation would be required if another mapping format was used.

Current CM-5 configurations have power-of-two numbers of processing nodes. In this case, the RSB algorithm is based on an iterative partitioning process which decomposes the whole mesh into 2 partitions, each of which in turn is decomposed into 2 partitions, and so on. However, future CM-5 systems may have non a power-of-two number of vector units. Therefore, the notion of "bisection" has to be extended by decomposing the number of vector units into a product of primes, viz.,

$$n_{\mathrm{vu}} = p_1 \times p_2 \times \ldots \times p_{N_{\mathrm{iter}}} \tag{1}$$

The partitioning algorithm is then applied recursively for $N_{\mathrm{iter}}$ iterations, with the subdomains obtained at iteration $i-1$ being decomposed into $p_i$ partitions.

The implementation of the algorithm is done such that all elements of the mesh are treated in parallel. It implies a two-level parallelization; one level on the partitions generated at a given stage of the recursive process and the other on the elements in each partition. This is to be compared with a serial implementation of the algorithm where not only the elements but also the partitions are processed sequentially. The number of calls to the partitioning process is therefore reduced from $n_{\mathrm{par}} - 1$ (in the sequential case) to at most $\lfloor \log_2(n_{\mathrm{vu}}) \rfloor$. Moreover, there is no loss of performance during the recursive process since the CM-5 system always processes the same number of data, namely the number of elements in the whole mesh.

The array `idual` is used to evaluate the Laplacian matrix $\boldsymbol{L}$, defined as

$$L_{ij} = \begin{cases} -1, & \text{if elements } i \text{ and } j \text{ share a face;} \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

$$L_{ii} = -\sum_{\substack{j=1 \\ j \neq i}}^{n_{\mathrm{el}}} L_{ij} \tag{3}$$

$\boldsymbol{L}$ is a positive semi-definite matrix. It can be easily shown that the eigenvector associated with the zero eigenvalue is $\boldsymbol{e} = \{1, 1, \ldots, 1\}^T$. The zero eigenvalue has a multiplicity equal to the number of connected element blocks in the mesh (or in the considered partition obtained at a given stage of the recursive process). By definition, a partition is said to be connected if the graph defined by `idual` for that partition is connected. The properties of the smallest non-zero eigenvalue and its associated eigenvector $\boldsymbol{f}$ have been studied by Fiedler in the framework of graph theory [5, 6, 7]. He has shown that reordering the components of $\boldsymbol{f}$ provides a reordering of the elements in the mesh (or in the corresponding partition). The reordered list of elements is then split as desired. The vector $\boldsymbol{f}$ will be referred to as the *Fiedler vector*.

Box 1 - Parallel Recursive Spectral Bisection Algorithm.

Given $n_{\rm el}$, $n_{\rm vu}$ and `idual`, proceed as follows:

*(Calculate the number of elements per partition)*
$\eta_{\rm el} = \lceil n_{\rm el}/n_{\rm vu} \rceil$
*(Decompose $n_{\rm vu}$ into a product of primes)*
$n_{\rm vu} = p_1 \times p_2 \times \ldots \times p_{N_{\rm iter}}$
*(Initialization)*
$\boldsymbol{d} = \{1, 0, \ldots, 0\}^T$ *(partition delimiter)*
$\boldsymbol{P} = \boldsymbol{I}_{n_{\rm el}}$ *(permutation mapping)*
$\bar{\eta}_{\rm el} = \eta_{\rm el}\, n_{\rm vu}$ *(current number of elements per partition)*
*(Loop over number of iterations)*
**Do** $n = 1, \ldots, N_{\rm iter}$
Identify the connected element blocks (see Box 2)
Calculate the Fiedler vector $\boldsymbol{f}$ (see Box 3)
Sort the components of $\boldsymbol{f}$ for each partition
Update $\boldsymbol{P}$
$\bar{\eta}_{\rm el} = \bar{\eta}_{\rm el}/p_{\boldsymbol{n}}$
$d_{i \times \bar{\eta}_{\rm el}+1} = 1 \qquad i = 0, 1, \ldots, \left\lfloor \dfrac{n_{\rm el} - 1}{\bar{\eta}_{\rm el}} \right\rfloor$
Reorder `idual` based on $\boldsymbol{P}$
Set `idual` entry to 0 for elements having a neighbor in a different partition
**End do**
**Return**

In the data-parallel implementation of the RSB algorithm, the smallest non-zero eigenvalue and the Fiedler vector are evaluated using a modified version of the Lanczos algorithm: The unnecessary computation of the zero eigenvalue is avoided by orthogonalizing all Lanczos vectors against $\boldsymbol{e}$, the eigenvector corresponding to the zero eigenvalue. Moreover, the smallest non-zero eigenvalues of the tridiagonal matrices generated by the Lanczos algorithm are computed using a modified method of bisection . The complete parallel RSB algorithm is presented in Box 1. The partition delimiter $\boldsymbol{d}$ defined in Box 1 is used to identify the first element in each partition. The following sections describe in greater details the important issues arising in the implementation of the parallel RSB algorithm on the CM-5 system.

Box 2 - Identification of connected element blocks.

Given `idual`, proceed as follows:

*(Initialization)*
`color` $= 0$
Mark all elements as non-colored
*(Element coloring)*
**Do while** elements remain to be colored
    `color` $=$ `color` $+ 1$
    Find the first non-colored element in each partition
    Assign `color` to these elements and mark them as colored
    Mark their neighbors as receivers of the color
    *(Neighbor coloring)*
    **Do while** neighbors to be colored still exist
        Send `color` to neighbors and mark them as colored
        Mark the non-colored neighbors' neighbors as receivers
    **End do**
**End do**
Reorder elements in each partition based on their color
Define the delimiter $\boldsymbol{d}'$ as

$$d'_i = \begin{cases} 1, & \text{if element } i \text{ is the first element in a connected block;} \\ 0, & \text{otherwise.} \end{cases}$$

**Return**

### 3.2. Identification of connected element blocks

Extending Fiedler's work to mesh partitioning shows that a connected partition is guaranteed to be decomposed into two connected subdomains only if the reordered components of the Fiedler vector are split according to their sign, i.e., negative components are associated with the first subdomain and positive components with the second subdomain. Unfortunately, the block distribution format imposes a split which may not yield connected partitions. If such a case occurs, the orthogonalization of the Lanczos vectors against $\boldsymbol{e}$ has to be done for each connected block of elements independently of the others. We therefore have to design a coloring algorithm which identifies the connected element blocks. A "snow ball" algorithm is presented in Box 2. In this algorithm, one element in each partition sends its color (initially set to 1) to its neighbors, which in turn send the color to their neighbors, and so on. The color is incremented and the algorithm restarted if there are no non-colored neighbors left. The algorithm terminates when all elements are assigned a color. The low latency of the data network makes the CM-5 system suitable for such algorithms, especially in the initial phase of the iterative process where only a small number of data are sent through the network.

### 3.3. Lanczos algorithm

The Lanczos algorithm used to compute the Fiedler vector is presented in Box 3. All array operations are performed component by component, as defined in the CM Fortran language. For example, the component-wise calculation

$$\boldsymbol{r}_i = \boldsymbol{u}_i - \boldsymbol{\alpha}_i \boldsymbol{v}_i \tag{4}$$

actually reads

$$r_{ik} = u_{ik} - \alpha_{ik} v_{ik} \qquad k = 1, \ldots, n_{\mathrm{el}} \tag{5}$$

The operation $\bar{\boldsymbol{z}} = \mathcal{E}(\boldsymbol{z})$ extracts the first component of $\boldsymbol{z}$ stored in the memory of each vector unit and copies it into an array $\bar{\boldsymbol{z}}$ of dimension $n_{\mathrm{vu}}$. For example, if $k$ components of $\boldsymbol{z}$ are stored on each vector unit, we have

$$\bar{z}_i = z_{(i-1)k+1} \qquad i = 1, \ldots, n_{\mathrm{vu}} \tag{6}$$

Note that this extraction happens without any need for communication.

The operation $\boldsymbol{z} = \mathcal{S}(\bar{\boldsymbol{z}}; \bar{\boldsymbol{d}})$ is a segmented scan-copy operation (available from the CMF Utility Library [31]) of $\bar{\boldsymbol{z}}$ using $\bar{\boldsymbol{d}}$ as the segment delimiter, followed by an on-vector

unit spread operation. In the hypothetical case where $n_{\mathrm{vu}} = 4$, $\boldsymbol{z}$ has 8 components (2 per vector unit),

$$\bar{\boldsymbol{z}} = \left\{ \begin{array}{c} \bar{z}_1 \\ \bar{z}_2 \\ \bar{z}_3 \\ \bar{z}_4 \end{array} \right\} \quad \text{and} \quad \bar{\boldsymbol{d}} = \left\{ \begin{array}{c} 1 \\ 0 \\ 1 \\ 0 \end{array} \right\}, \tag{7}$$

the segmented scan-copy operation gives

$$\text{scan-copy}(\bar{\boldsymbol{z}}; \bar{\boldsymbol{d}}) = \left\{ \begin{array}{c} \bar{z}_1 \\ \bar{z}_1 \\ \bar{z}_3 \\ \bar{z}_3 \end{array} \right\} \tag{8}$$

Since two components of $\boldsymbol{z}$ are stored on each vector unit, the spread operation can be written

$$z_{2(i-1)+j} = \bar{z}_i \qquad i = 1, \ldots, n_{\mathrm{vu}} \qquad j = 1, 2 \tag{9}$$

which yields the final result

$$\boldsymbol{z} = \left\{ \begin{array}{c} \bar{z}_1 \\ \bar{z}_1 \\ \bar{z}_1 \\ \bar{z}_1 \\ \bar{z}_3 \\ \bar{z}_3 \\ \bar{z}_3 \\ \bar{z}_3 \end{array} \right\} \tag{10}$$

The concurrent dot-products, the matrix-vector products and the eigenvalue analysis are described in the following sections.

Box 3 - Computation of the Fiedler vector.

Given $\boldsymbol{L}$, $\boldsymbol{d}$, $\boldsymbol{d'}$, $n_{\rm vu}$, $\bar{\eta}_{\rm el}$, $N_{\max}$ and $\varepsilon_{\rm tol}$, proceed as follows:

*(Initialization)*

$\bar{\boldsymbol{d}} = \mathcal{E}(\boldsymbol{d}) = \{\bar{d}_j\}_{1 \le j \le \boldsymbol{n}_{\rm vu}}$

$\boldsymbol{e} = \{1, 1, \ldots, 1\}^T$

$\boldsymbol{v}_1 = \{1, 2, \ldots, \bar{\eta}_{\rm el}, 1, 2, \ldots, \bar{\eta}_{\rm el}, 1, \ldots\}^T$

$$\boldsymbol{v}_1 = \boldsymbol{v}_1 - \frac{(\boldsymbol{v}_1, \boldsymbol{e})_{\boldsymbol{d'}}}{(\boldsymbol{e}, \boldsymbol{e})_{\boldsymbol{d'}}} \boldsymbol{e}$$

$$\boldsymbol{v}_1 = \boldsymbol{v}_1 / (\boldsymbol{v}_1, \boldsymbol{v}_1)_{\boldsymbol{d}}^{1/2}$$

$\boldsymbol{u}_1 = \boldsymbol{L}\, \boldsymbol{v}_1$

*(Loop over Lanczos iterations)*

**Do** $i = 1, \ldots, N_{\max}$

$i_{\max} = i$

*(First part of Lanczos)*

$\boldsymbol{\alpha}_i = (\boldsymbol{u}_i, \boldsymbol{v}_i)_{\boldsymbol{d}}$

$\boldsymbol{r}_i = \boldsymbol{u}_i - \boldsymbol{\alpha}_i\, \boldsymbol{v}_i$

$$\boldsymbol{r}_i = \boldsymbol{r}_i - \frac{(\boldsymbol{r}_i, \boldsymbol{e})_{\boldsymbol{d'}}}{(\boldsymbol{e}, \boldsymbol{e})_{\boldsymbol{d'}}} \boldsymbol{e}$$

$\boldsymbol{\beta}_i = (\boldsymbol{r}_i, \boldsymbol{r}_i)_{\boldsymbol{d}}^{1/2}$

*(Eigenvalue analysis)*

$\bar{\boldsymbol{\alpha}}_i = \mathcal{E}(\boldsymbol{\alpha}_i) = \{\bar{\alpha}_{i,j}\}_{1 \le j \le \boldsymbol{n}_{\rm vu}}$

$\bar{\boldsymbol{\beta}}_i = \mathcal{E}(\boldsymbol{\beta}_i) = \{\bar{\beta}_{i,j}\}_{1 \le j \le \boldsymbol{n}_{\rm vu}}$

**Where** $\bar{d}_j = 1$, calculate the eigenvector $\boldsymbol{\zeta}_j^{(i)} = \{\zeta_{k,j}\}_{1 \le k \le i}$ associated with the largest eigenvalue $\lambda_j^{(i)}$ of

$$\boldsymbol{T}_j^{(i)} = \begin{bmatrix} \bar{\alpha}_{1,j} & \bar{\beta}_{1,j} & & 0 \\ \bar{\beta}_{1,j} & \bar{\alpha}_{2,j} & \ddots & \\ & \ddots & \ddots & \bar{\beta}_{i-1,j} \\ 0 & & \bar{\beta}_{i-1,j} & \bar{\alpha}_{i,j} \end{bmatrix}, \quad j = 1, \ldots, n_{\rm vu} \qquad \text{(see Box 4)}$$

**If** accuracy of all $\lambda_j^{(i)}$'s is within $[0, \varepsilon_{\rm tol}]$, **Exit** $i$ loop

*(Second part of Lanczos)*

$\boldsymbol{v}_{i+1} = \boldsymbol{r}_i / \boldsymbol{\beta}_i$

$\boldsymbol{u}_{i+1} = \boldsymbol{L}\, \boldsymbol{v}_{i+1} - \boldsymbol{\beta}_i\, \boldsymbol{v}_i$

**End do**

*(Calculate the Fiedler vector)*

$\bar{\boldsymbol{\zeta}}_i = \{\zeta_{i,j}\}_{1 \le j \le \boldsymbol{n}_{\rm vu}} \qquad i = 1, \ldots, i_{\max}$

$$\boldsymbol{f} = \sum_{i=i_{\max}}^{1} \mathcal{S}(\bar{\boldsymbol{\zeta}}_i; \bar{\boldsymbol{d}})\, \boldsymbol{v}_i$$

*(End)*

**Return**

### 3.3.1. Concurrent dot-products

The operator $(\cdot,\cdot)_{\boldsymbol{\delta}}$ indicates a dot-product on each partition defined by the delimiter $\boldsymbol{\delta}$. Consider the case of two partitions:

$$\boldsymbol{u} = \left\{ \begin{array}{c} u_1 \\ u_2 \\ \vdots \\ u_{\boldsymbol{p}-1} \\ u_{\boldsymbol{p}} \\ u_{\boldsymbol{p}+1} \\ \vdots \\ u_{\boldsymbol{n}_{\rm el}} \end{array} \right\} \quad , \quad \boldsymbol{v} = \left\{ \begin{array}{c} v_1 \\ v_2 \\ \vdots \\ v_{\boldsymbol{p}-1} \\ v_{\boldsymbol{p}} \\ v_{\boldsymbol{p}+1} \\ \vdots \\ v_{\boldsymbol{n}_{\rm el}} \end{array} \right\} \quad \text{and} \quad \boldsymbol{\delta} = \left\{ \begin{array}{c} 1 \\ 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{array} \right\} \tag{11}$$

Then,

$$(\boldsymbol{u}, \boldsymbol{v})_{\boldsymbol{\delta}} = \left\{ \begin{array}{c} u_1 v_1 + \ldots + u_{\boldsymbol{p}-1} v_{\boldsymbol{p}-1} \\ u_1 v_1 + \ldots + u_{\boldsymbol{p}-1} v_{\boldsymbol{p}-1} \\ \vdots \\ u_1 v_1 + \ldots + u_{\boldsymbol{p}-1} v_{\boldsymbol{p}-1} \\ u_{\boldsymbol{p}} v_{\boldsymbol{p}} + \ldots + u_{\boldsymbol{n}_{\rm el}} v_{\boldsymbol{n}_{\rm el}} \\ u_{\boldsymbol{p}} v_{\boldsymbol{p}} + \ldots + u_{\boldsymbol{n}_{\rm el}} v_{\boldsymbol{n}_{\rm el}} \\ \vdots \\ u_{\boldsymbol{p}} v_{\boldsymbol{p}} + \ldots + u_{\boldsymbol{n}_{\rm el}} v_{\boldsymbol{n}_{\rm el}} \end{array} \right\} \tag{12}$$

This operation is actually performed in three steps:

1. A component-wise product of $\boldsymbol{u}$ and $\boldsymbol{v}$ is performed, i.e.,

$$\boldsymbol{u}\,\boldsymbol{v} = \left\{ \begin{array}{c} u_1 v_1 \\ \vdots \\ u_{\boldsymbol{n}_{\rm el}} v_{\boldsymbol{n}_{\rm el}} \end{array} \right\} \tag{13}$$

2. A reversed segmented scan-add operation is executed using $\boldsymbol{\delta}$ as the delimiter:

$$\text{scan-add}(\boldsymbol{u}\,\boldsymbol{v};\boldsymbol{\delta}) = \left\{ \begin{array}{c} u_1 v_1 + u_2 v_2 + \ldots + u_{p-1} v_{p-1} \\ u_2 v_2 + \ldots + u_{p-1} v_{p-1} \\ \vdots \\ u_{p-1} v_{p-1} \\ u_p v_p + u_{p+1} v_{p+1} + \ldots + u_{n_{\text{el}}} v_{n_{\text{el}}} \\ u_{p+1} v_{p+1} + \ldots + u_{n_{\text{el}}} v_{n_{\text{el}}} \\ \vdots \\ u_{n_{\text{el}}} v_{n_{\text{el}}} \end{array} \right\} \tag{14}$$

3. A segmented scan-copy operation is performed on the result of Step 2, leading to $(\boldsymbol{u},\boldsymbol{v})_{\boldsymbol{\delta}}$.

The procedure described above is very general and can perform concurrent dot-products using any delimiter $\boldsymbol{\delta}$. However, when $\boldsymbol{\delta} = \boldsymbol{d}$, or when $\boldsymbol{\delta} = \boldsymbol{d}'$ and all partitions are connected (which effectively means $\boldsymbol{d}' = \boldsymbol{d}$), the following more efficient strategy is used: Since the partitioning strictly follows the block distribution format of the CM-5 run-time system, the first element of each partition is also the first element stored in the memory of a vector unit. Consequently, the operation $(\boldsymbol{u},\boldsymbol{v})_{\boldsymbol{d}}$ is performed in the following steps:

1. Perform the component-wise product $\boldsymbol{u}\,\boldsymbol{v}$.
2. Perform on-vector unit reductions-with-addition of $\boldsymbol{u}\,\boldsymbol{v}$.
3. Scatter the results of Step 2 to the vector units containing the first element of each partition.
4. Gather the results of Step 3 to each vector unit containing elements of the same partition.
5. Perform an on-vector unit spread of the results of Step 4, which yields $(\boldsymbol{u},\boldsymbol{v})_{\boldsymbol{d}}$.

The on-vector unit reduction and spread operations are executed through calls to run-time system functions. The gather and scatter operations are achieved through the CMSSL communication primitives `sparse_util_gather` and `sparse_util_scatter` [32]. A description of the methodology implemented in these communication routines along with performance results is given in [18].

### 3.3.2. Matrix-vector products

The most expensive operations in the evaluation of the Fiedler vector are the matrix-vector products of the form $\boldsymbol{u} = \boldsymbol{L}\,\boldsymbol{v}$. They have to be handled with special care to achieve

good performance. The matrix-vector product $\boldsymbol{u} = \boldsymbol{L}\,\boldsymbol{v}$ can be decomposed into two parts:

$$u_k = L_{kk} v_k + \sum_{\substack{l=1 \\ l \neq k}}^{n_{\mathrm{el}}} L_{kl} v_l \qquad k = 1, \ldots, n_{\mathrm{el}} \tag{15}$$

The first term is simply a component-wise product between the vectors diag($\boldsymbol{L}$) and $\boldsymbol{v}$ and does not require any communication between processing nodes. Since $L_{kl} = 0$ or $-1$ for all $k \neq l$, the second term is actually a scatter operation. It can be rewritten

$$\sum_{\substack{l=1 \\ l \neq k}}^{n_{\mathrm{el}}} L_{kl} v_l = - \sum_{\substack{l=1 \\ \mathtt{idual}(l,k) \neq 0}}^{n_{\mathrm{faces}}} v_{\mathtt{idual}(l,k)} \qquad k = 1, \ldots, n_{\mathrm{el}} \tag{16}$$

The scatter operation is achieved through the communication primitive `sparse_util_scatter`. A mask is passed to this routine to allow the scatter only for the non-zero entries of `idual`.

### 3.3.3. Eigenvalue analysis

The computation of the smallest non-zero eigenvalues $\{\lambda_j^{(i)}\}_{1 \leq j \leq n_{\mathrm{vu}}}$ is done using the modified method of bisection presented in Box 4. Following the Sturm sequence property [8], the initial bisection intervals are $[0, \lambda_j^{(i-1)}]$ for $i \geq 3$, $\lambda_j^{(1)}$ and $\lambda_j^{(2)}$ being explicitly calculated. One can note that the eigenvalues $\lambda_j^{(i)}$ converge towards the upper bounds of the initial bisection intervals as the Lanczos iteration number $i$ increases. This has led us to modify the classical method of bisection by performing an unequal split of the bisection interval to achieve faster convergence of the eigenvalue analysis. The method reverts itself to equal splits when the upper bounds of the intervals are changed. Numerical experiments demonstrates that a 3-to-1 split generated improvements over a 1-to-1 split in a consistent way. Such a strategy has shown performance comparable to the algorithm described in [20]. In addition, one should note that there will be at most one eigenvalue computation per vector unit. The modified method of bisection has therefore been implemented in CDPEAC [29] (a macro-assembler) to get the best possible performance in scalar mode.

## 3.4. Remarks

A few additional remarks can be made about the parallel RSB algorithm:

1. The first Lanczos vector $\boldsymbol{v}_1$ (in Box 3) is chosen such that information on the initial element ordering (or on the element reordering from the previous iteration of

the recursive process) is passed to the Lanczos algorithm. Some mesh generators construct meshes in an orderly fashion which may help the Lanczos algorithm to converge faster.

2. Elements are sorted after each identification of connected blocks and after each evaluation of the Fiedler vector. An efficient sorting routine is therefore mandatory to achieve good overall performance. Our implementation uses the fast sorting routines available in the CMSSL.

Box 4 - Modified Method of Bisection.

Given $\boldsymbol{T}_j^{(i)}$, $\lambda_j^{(i-1)}$ and $\varepsilon_{\text{tol}}$, proceed as follows:

$\lambda_j^{\min} = 0$
$\lambda_j^{\max} = \lambda_j^{(i-1)}$
$\gamma_j = 3/4$
$\bar{\beta}_{0,j} = 0$
**Repeat**
    $\lambda_j = (1-\gamma_j)\lambda_j^{\min} + \gamma_j \lambda_j^{\max}$
    $\delta_j = 1$
    $\sigma_j = 0$
    **Do** $k = 1, \ldots, i$
        $\delta_j = \bar{\alpha}_{k,j} - \lambda_j - \bar{\beta}_{k-1,j}^2 / \delta_j$
        **If** $(\delta_j < 0)$ $\sigma_j = \sigma_j + 1$
    **End do**
    **If** $(\sigma_j \geq 1)$
        $\lambda_j^{\max} = \lambda_j$
        $\gamma_j = 1/2$
    **Else**
        $\lambda_j^{\min} = \lambda_j$
    **End if**
**Until** $\lambda_j^{\max} - \lambda_j^{\min} \leq \varepsilon_{\text{tol}}\, \lambda_j$
$\lambda_j^{(i)} = \lambda_j^{\max}$
**Return**

### 3.5. Partitioning examples

We present two numerical examples which demonstrate the performance of the parallel RSB algorithm and the quality of the partitions it generates. The partitioning code was compiled with CMF 2.1 Beta 0.1 and was run on a timeshared 32-node CM-5 system equipped with 128 vector units. This CM-5 system was running the Connection Machine operating system CMOST 7.2 Beta 1.1. All reported timings correspond to CM-busy times (which represent a measure of CPU times on the processing nodes).

#### 3.5.1. Cylindrical leading edge

This first example uses an adaptively refined tetrahedral mesh having 16,707 nodes and 86,701 elements (see Figure 2). This mesh was used by Thareja *et al.* [26] to compute the three-dimensional inviscid shock-shock interaction on a swept cylindrical leading edge. This mesh is first used to study the variations in the partitioning as a function of the Lanczos tolerance $\varepsilon_{\text{tol}}$. For this purpose solely, we measure the quality of the partitioning by computing the number of faces shared by two subdomains (also referred to as the number of cuts [9, 24]). The parallel RSB algorithm was run on a 32-node CM-5 system for tolerance values ranging from $10^{-8}$ to $10^{-1}$. The maximum number of Lanczos iterations $N_{\max}$ is set to a high value (500 in this case). Table 1 presents the number of cuts and the time spent partitioning the mesh as a function of $\varepsilon_{\text{tol}}$. One can see that the number of cuts remains stable for $\varepsilon_{\text{tol}} \leq 10^{-3}$, but the partitioning time increases as the tolerance is tightened. The high partitioning time for $\varepsilon_{\text{tol}} = 10^{-1}$ is caused by the generation of a large number of disconnected element blocks as the bisection process progresses (up to 166 blocks). Such a large number of blocks slows down the evaluation of dot-products in the Lanczos algorithm. In light of this example and other empirical analyses not presented here, we have chosen to set $\varepsilon_{\text{tol}} = 10^{-4}$ and $N_{\max} = 300$ for all numerical examples presented in the remainder of this paper.

*Table 1.* Cylindrical leading edge. Number of cuts and CM-busy time for different values of the Lanczos tolerance $\varepsilon_{\text{tol}}$ on a 32-node CM-5 system.

| $\varepsilon_{\text{tol}}$ | $10^{-1}$ | $10^{-2}$ | $10^{-3}$ | $10^{-4}$ | $10^{-5}$ | $10^{-6}$ | $10^{-7}$ | $10^{-8}$ |
|---|---|---|---|---|---|---|---|---|
| No. of cuts | 40,456 | 20,820 | 18,062 | 18,079 | 17,883 | 18,037 | 18,001 | 18,043 |
| Timings | 76.7 s | 38.4 s | 39.2 s | 50.7 s | 64.8 s | 70.7 s | 81.6 s | 102.3 s |

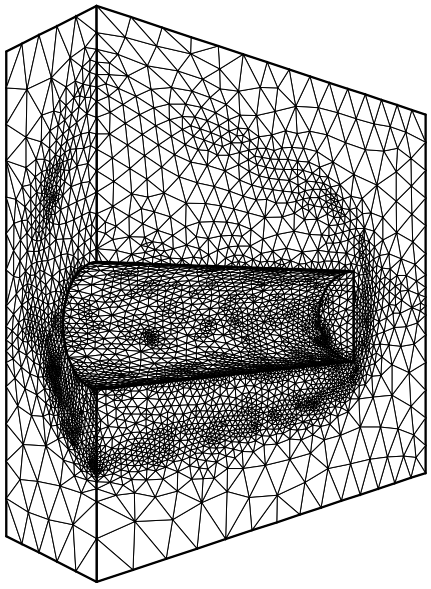

*Figure 2.* Cylindrical leading edge. View of the surface mesh.

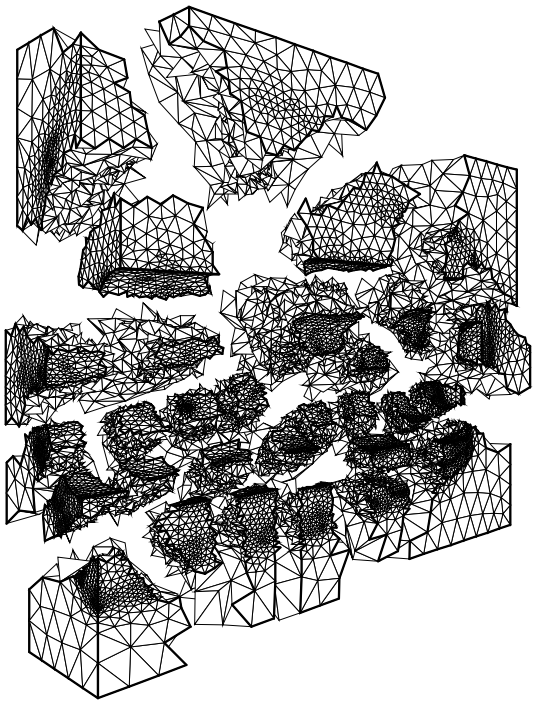

*Figure 3.* Cylindrical leading edge. Decomposition into 32 partitions.

The second part of this study involves a computing cost analysis. Using the values of $\varepsilon_{\text{tol}}$ and $N_{\max}$ just mentioned, the partitioning into 32 subdomains is shown in Figure 3. The quality of the decomposition is particularly striking, even in the adaptively refined regions. This is due in part to the fact that the partitioning is based solely on topological information (the dual mesh connectivity) and is not function of any geometric property. Note that 128 partitions are actually needed on a 32-node CM-5 system, but the resulting picture is too confusing to be shown here. However, all timings presented are for a partitioning into 128 subdomains.

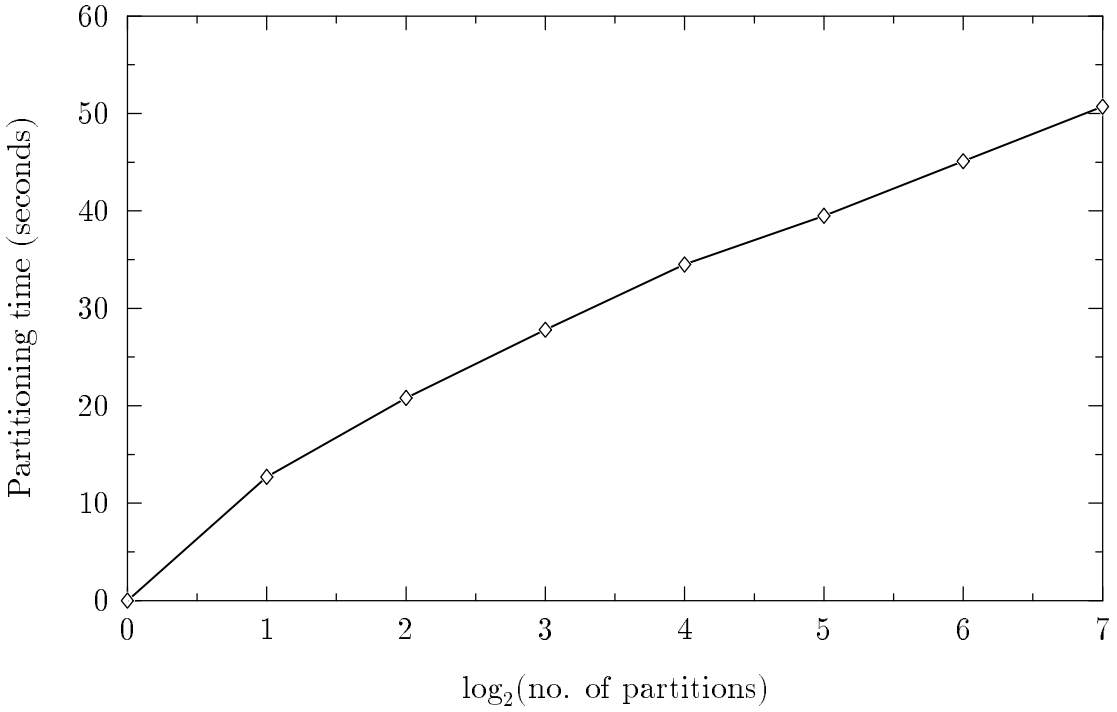


*Figure 4.* Cylindrical leading edge. Partitioning cost as a function of recursive bisection on a 32-node CM-5 system.

Figure 4 depicts the cost of the parallel RSB algorithm as the bisection procedure progresses. The slow increase in computing cost during the recursive bisection process results from two properties:

1. The two levels of parallelization in the algorithm (see Section 3.1) would lead to a cost of $O(\log_2(\text{no. of partitions}))$ if the number of Lanczos iterations per bisection was constant.
2. However, in reality, the number of Lanczos iterations decreases as the bisection pro-

cedure progresses since the size of the subdomains to be bisected gets smaller. This behavior leads to a cost better than $O(\log_2(\text{no. of partitions}))$, as seen in Figure 4. This sub-$\log_2(n)$ computing cost tends to indicate that the parallel RSB algorithm will be very efficient on massively parallel systems.

*Table 2.* Cylindrical leading edge. CM-busy times for different parts of the parallel RSB algorithm for a partitioning into 128 subdomains on a 32-node CM-5 system.

| | Timings | Percentage |
|---|---|---|
| ident. of connected blocks | 6.4 s | 12.6% |
| comp. of Fiedler vector | 40.5 s | 79.9% |
| data sorting/reordering | 2.2 s | 4.3% |
| miscellaneous | 1.6 s | 3.2% |
| Total | 50.7 s | 100.0% |

*Table 3.* Cylindrical leading edge. Detailed cost analysis for the computation of the Fiedler vector.

| | Timings | Percentage |
|---|---|---|
| matrix-vector products | 23.9 s | 59.0% |
| dot-products | 8.8 s | 21.7% |
| eigenvalue analyses | 3.6 s | 8.9% |
| SAXPYs and miscellaneous | 4.2 s | 10.4% |
| Total | 40.5 s | 100.0% |

The overall cost of the partitioning algorithm for this example is 50.7 seconds. Table 2 shows the computing costs of the different parts of the partitioning algorithm. The computation of the Fiedler vector using the Lanczos algorithm dominates with 80% of the total time. A more detailed cost analysis of the Lanczos algorithm is presented in Table 3. One can deduct from these two tables that 85% of the total time is spent in communication between processing nodes (the communication-dominated portions of the code are the identification of connected blocks, matrix-vector products, dot-products, sorting and reordering). Nonetheless, the parallel RSB algorithm exhibits good performance on the CM-5 system.

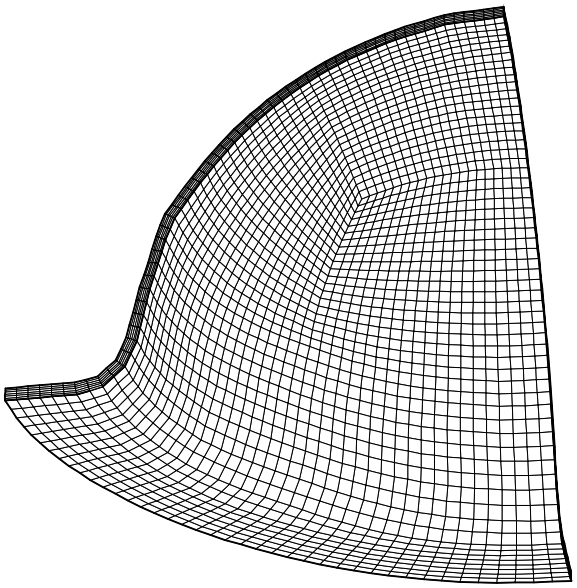

*Figure 5.* Aluminum sheet. View of the surface mesh.

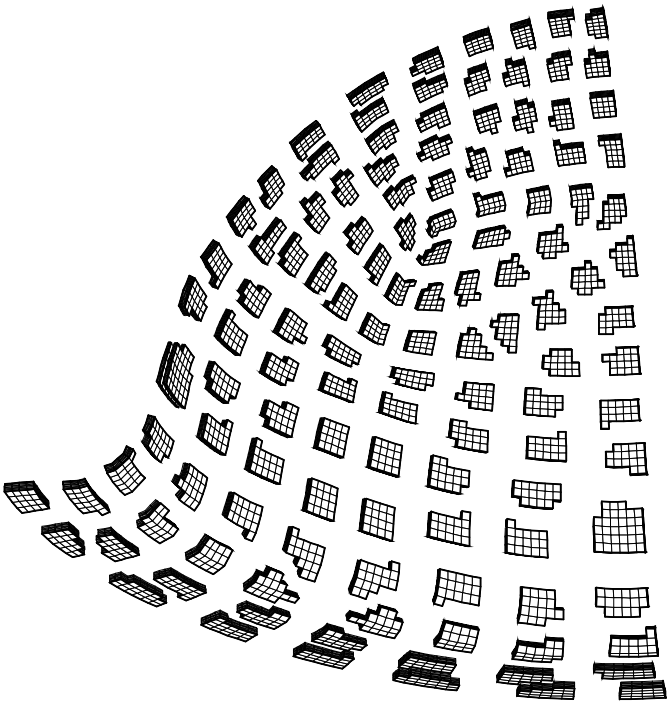

*Figure 6.* Aluminum sheet. Decomposition for a 32-node CM-5 system.

### 3.5.2. Deformed aluminum sheet

This example demonstrates the ability of the parallel RSB algorithm to partition meshes made of any element type. The mesh shown in Figure 5 has 14,567 nodes and 12,000 tri-linear bricks. It has been used by Beaudoin *et al.* [2] to study hydroforming of an aluminum sheet. The partitioning for a 32-node CM-5 system is depicted in Figure 6. A good quality in the decomposition is also obtained for this example, even though the partitioning is not optimal given the structuredness of the mesh. A smaller tolerance $\varepsilon_{\text{tol}}$ would have improved the quality of the partitioning, but at a higher cost. There is a trade-off between cost and quality of partitioning, and the decomposition shown in Figure 6 is sufficient for all practical purposes. The partitioning cost as a function of the recursive bisection process is presented in Figure 7. The same cost-related behavior as in the previous example can be observed.

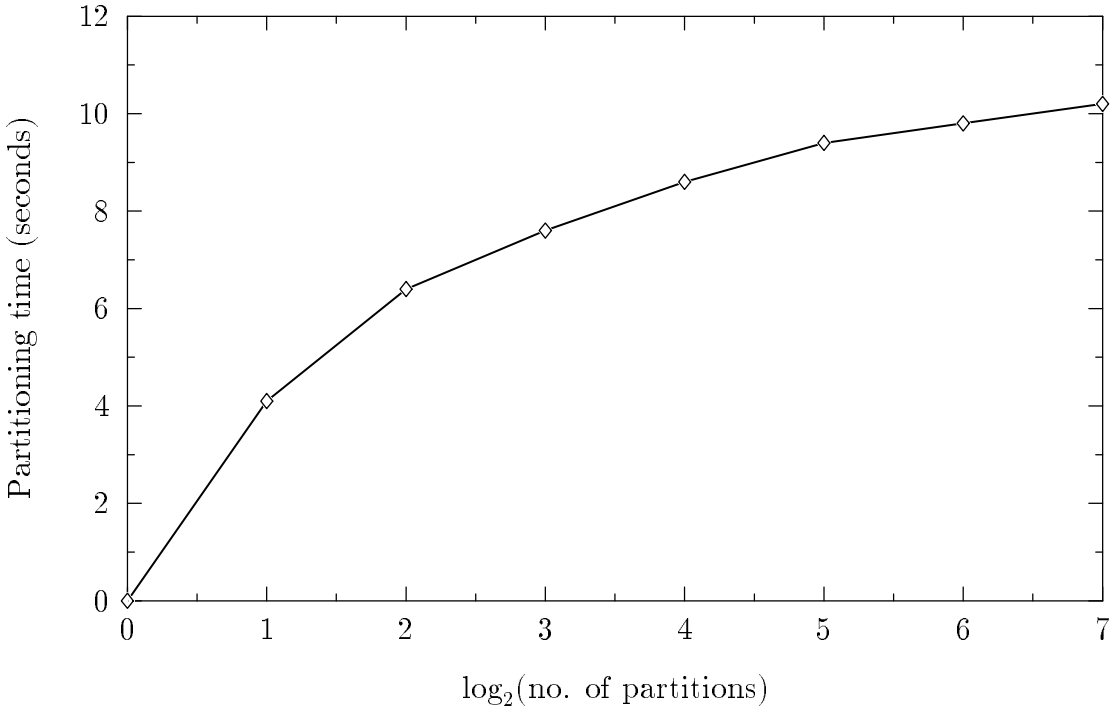


*Figure 7.* Aluminum sheet. Partitioning cost as a function of recursive bisection on a 32-node CM-5 system.

## 4. Node renumbering algorithm

The second step in the data mapping strategy is to renumber the mesh nodes such that most nodes associated with an element partition reside on the same vector unit as that partition.

The relationship between elements and nodes is usually expressed through the *element nodes array* `ien` of dimension $n_{\mathrm{en}} \times n_{\mathrm{el}}$ which contains the list of nodes attached to a given element. $n_{\mathrm{en}}$ is the number of element nodes (e.g., $n_{\mathrm{en}} = 4$ for a linear tetrahedron). A heuristic algorithm implemented in CM Fortran performs the node renumbering in two passes: In the first pass, the nodes interior to the element partitions, i.e., nodes which are associated with elements of *only* one partition, are identified and placed on the proper vector units, with the block distribution restriction of having at most $\lceil n_{\mathrm{np}}/n_{\mathrm{vu}} \rceil$ nodes per vector unit. $n_{\mathrm{np}}$ is the total number of nodes in the mesh. This initial filling is performed using a scan-add operation followed by a one-to-one send of the node numbers to their corresponding vector unit. In the second pass, nodes not yet allocated are distributed among the vector units which have not been filled. This second filling is done through the CM Fortran intrinsics `PACK` and `UNPACK`. The complete algorithm is implemented in less than 40 lines of Fortran and executes in a matter of seconds for large meshes.

## 5. Communication primitives

Implementations of finite element solvers on parallel computers usually contain three major procedures:

1. Gather the current solution from the nodes to the elements.
2. Compute the element contributions to the residual of the variational formulation.
3. Assemble the element residuals by scattering them to the nodes.

These three procedures are often embedded in an iterative solver loop, or a time-stepping loop, or both. For details on data-parallel finite element techniques, see [4, 15]. On the CM-5 system, both gather and scatter operations require communication between processing nodes. The objective is to take advantage of the data mappings presented in the previous sections in order to reduce the total communication time.

The element partitions generated by the parallel RSB algorithm can be viewed as meshes independent of one another. Each vector unit has its own mesh with local element and node numberings. The gather operation is then performed in two steps:

1. A global gather operation is executed between the global set of nodes (i.e., the nodes for the whole mesh) and the local sets of nodes (i.e., the nodes for each partition). The CMSSL routine `sparse_util_gather` is used for this operation. This routine calls optimized functions of the run-time system which determine the type of motion required by each datum (on vector unit, off vector unit but on processing node, or off processing node) and use the appropriate hardware to perform the gather operation.
2. A local gather operation is then executed on each vector unit between the local sets of nodes and elements. The local gather requires *no* communication between processing nodes. It has been implemented in C with calls to low-level library routines to take full advantage of the indirect addressing available on the CM-5 vector units. Experiments on unstructured meshes have shown local gather rates of 25 Mbytes/s per vector unit.

The scatter operation is performed in a similar fashion by having a local scatter followed by a global scatter. In an initial phase to the local scatter operation, a coloring algorithm is applied to the elements of each partition in order to avoid collisions. The local scatter is then performed by an on-vector unit one-to-one mapping followed by a reduction operation. The reduction adds up values which have been sent to each local node by its surrounding elements. The global scatter operation is achieved through a call to the CMSSL routine `sparse_util_scatter`. These two-step communication primitives are summarized by the schematic shown in Figure 8. It should be noted that the generation of the local connectivities for each partition as well as the element coloring for the local scatter need to be done only once. These operations are therefore performed in a preprocessing phase before the actual call to the finite element solver.

The percentage of data which are processed by the global step (i.e., the step which *may* require communication between processing nodes) can be easily evaluated:

1. The number of data moved during a gather or a scatter operation is equal to

$$n_{\mathrm{dof}} \times n_{\mathrm{en}} \times n_{\mathrm{el}} \tag{17}$$

   where $n_{\mathrm{dof}}$ is the number of degrees of freedom per node. Note that (17) is also the number of data moved during the local gather and scatter operations.

2. The number of data processed by the global gather/scatter steps is equal to

$$n_{\mathrm{dof}} \times \sum_{i=1}^{n_{\mathrm{par}}} n_{\mathrm{np}}^{(i)} \tag{18}$$

   where $n_{\mathrm{np}}^{(i)}$ is the number of nodes in partition $i$.

The value of $\sum_{i=1}^{n_{\rm par}} n_{\rm np}^{(i)}$ depends on the number of partitions, the element type and the number of elements per partition. One can however note that this number converges to $n_{\rm np}$ for infinitely large partitions. As a practical example, we can use the cylindrical leading edge presented in Section 3.5.1. For this mesh partitioned for a 32-node CM-5 system, we have

$$n_{\rm en} \times n_{\rm el} = 346{,}804 \tag{19}$$

$$\sum_{i=1}^{n_{\rm par}} n_{\rm np}^{(i)} = 28{,}103 \tag{20}$$

which indicates that the number of data processed by the global step represents only 8.1% of the number of data actually gathered or scattered. Similarly, the hexahedral mesh presented in Section 3.5.2 yields 23.8 percentage points. This simple analysis also shows that the quality of a mesh partitioning can be measured by the value of $\sum_{i=1}^{n_{\rm par}} n_{\rm np}^{(i)}$: the smaller this value, the smaller the communication cost, and therefore the better the partitioning.

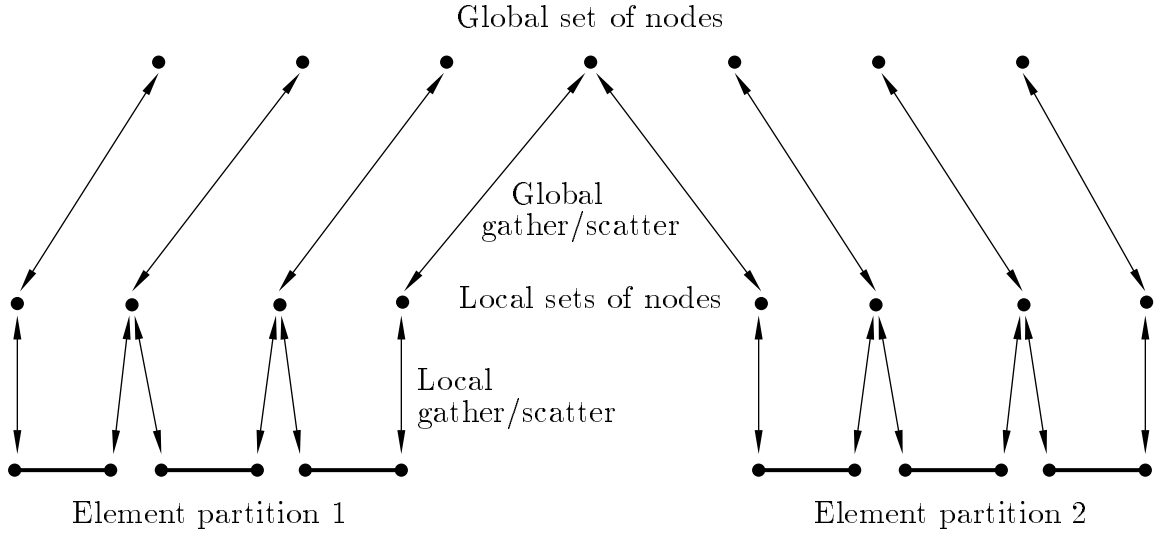


*Figure 8.* Pictorial description of the two-step gather/scatter operations.

## 6. Numerical examples

The fluid flow problems presented below have been solved using a data-parallel finite element program written in CM Fortran. The variational form is based on the

Galerkin/least-squares formulation [10, 11, 12, 13]. A matrix-free implicit iterative solver based on the GMRES algorithm is used to converge solutions to steady state [16, 22, 23]. The reader should refer to [14] and [15] for details related to the data-parallel implementation. The mesh partitioning algorithm and the communication primitives presented above have been used in the current version of the program. The tolerance and the Krylov space size in GMRES are set to 0.1 and 5, respectively. The GMRES solver restarts itself if needed. All computations are done in 64-bit arithmetic using only one integration point per element. The finite element program is compiled using CMF 2.1 Beta 0.1. Unless noted otherwise, all examples are run on timeshared CM-5 systems under CMOST 7.2 Beta 1.1 and all reported timings correspond to CM-busy times.

### 6.1. Falcon Jet

This example is used to study the degradation in parallel efficiency of the finite element program as the number of processing nodes in the CM-5 system increases. The inviscid flow at Mach 0.85 past a Falcon Jet is computed. The angle of attack is 1 degree. The mesh has 19,417 nodes and 109,914 linear tetrahedra (see Figure 9 for a view of the surface mesh on the airplane). 50 time steps at a CFL number of 10 are required to converge this problem to engineering accuracy (three orders of magnitude in residual decrease). The solver is initialized by a uniform flow.

This numerical example was run on 32-node, 64-node, 128-node, 256-node and 512-node CM-5 systems. We have chosen to report both CM-busy and elapsed times in order to better analyze performance degradation. The elapsed time is the sum of the CM-busy time and the idle time of the processing nodes (idling possibly caused by some computations being performed on the control processor). Note that the elapsed time ***is not*** equivalent to the wall-clock time. Timings and performance per processing node for the solver are presented in Figures 10 and 11, respectively. There appears to be a fairly constant overhead (i.e., difference between elapsed and CM-busy times) of about one minute for all the runs. Definite causes of this overhead are not currently known, but we have noticed several factors that may contribute to this problem:

1. Elapsed times (and CM-busy times to a minor extent) are very sensitive to the UNIX environment setup and the load of the CM-5 system (both on the control processor and on the processing nodes). Even though all timings were done on dedicated systems (only one job running at a time), there is still some overhead associated with the timesharing environment.

2. The current beta release of the CM Fortran compiler generates suboptimal code for the handling of array geometries, which may also contribute to the overall overhead.

All these issues are being analyzed and should be fixed in the final release of the CM-5 software. A very small difference between elapsed and CM-busy times is then to be expected, indicating that the control processor should not have any impact on the performance.

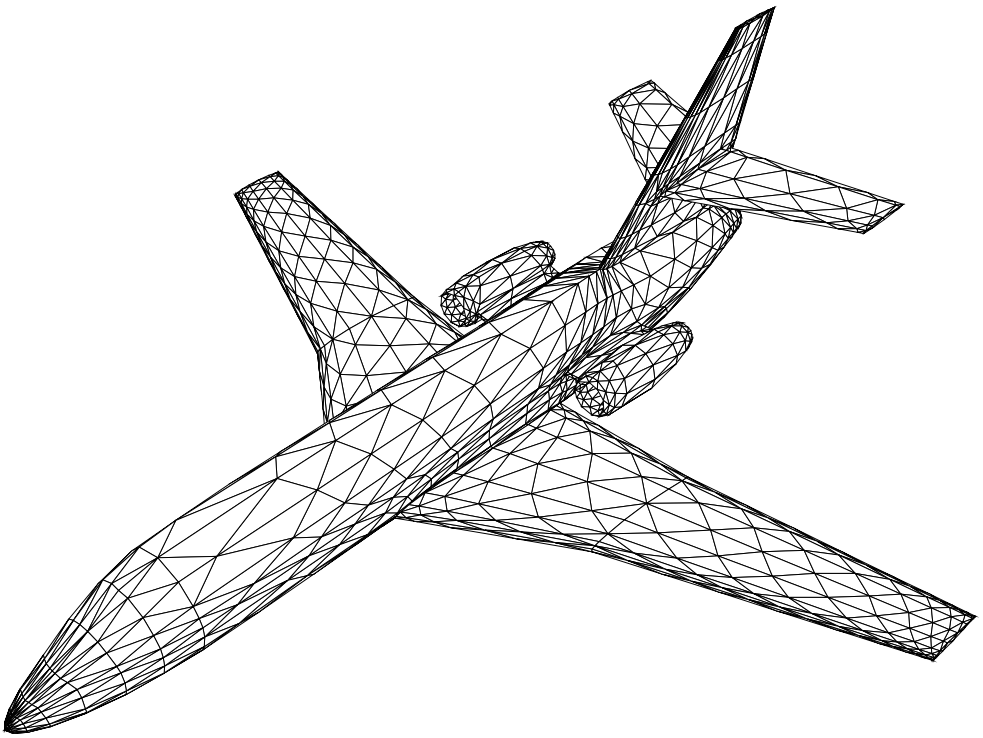

*Figure 9.* Falcon Jet. View of surface mesh on the airplane.

Figures 12 and 13 depict gather and scatter bandwidths as a function of the number of processing nodes. The ratio $\sum_{i=1}^{n_{\mathrm{par}}} n_{\mathrm{np}}^{(i)}/(n_{\mathrm{en}} \times n_{\mathrm{el}})$ (see Section 5) increases with the number of processing nodes, which explains the decrease in gather and scatter bandwidths (remember that the total bandwidth is a weighted average between memory and network bandwidths. For a fixed mesh size, the weight gets shifted from "memory" to "network" as the number of processing nodes increases). However, a factor of 4.3 for the gather and 3.1 for the scatter in bandwidth degradation is observed when the number of processing nodes is increased by a factor of 16, which indicates a good quality of the partitioning even for small meshes relative to the CM-5 system configuration.

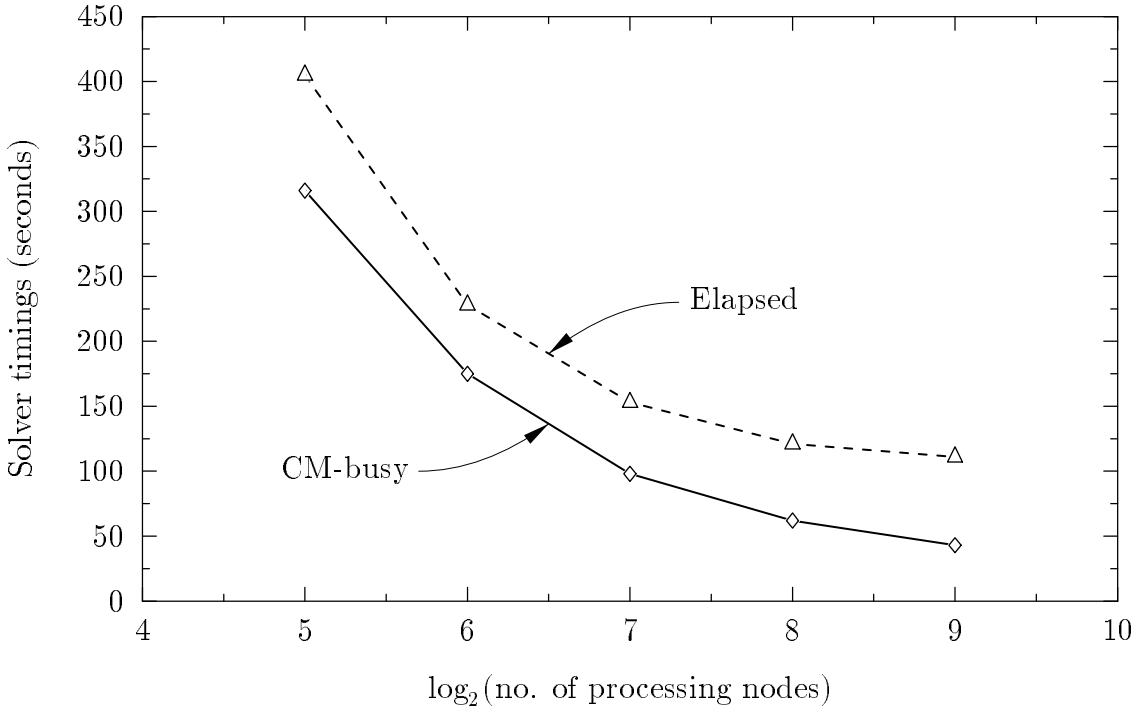


*Figure 10.* Falcon Jet. Solver timings as a function of the CM-5 configuration.

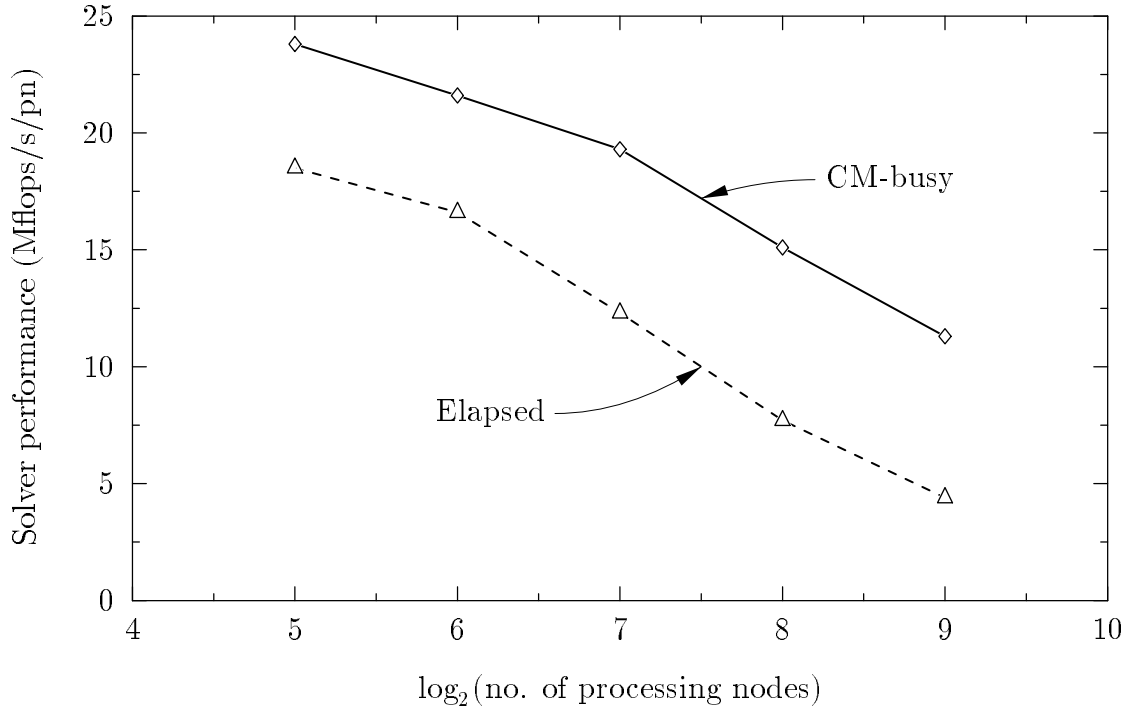


*Figure 11.* Falcon Jet. Solver performance per processing node as a function of the CM-5 configuration.

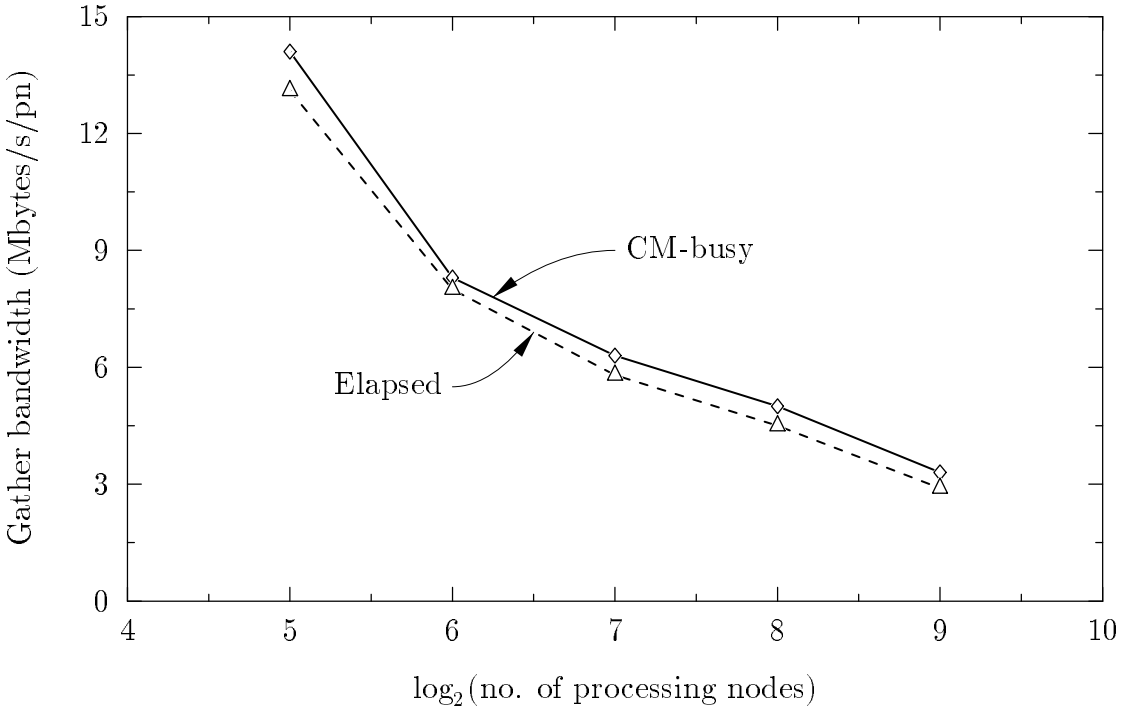


*Figure 12.* Falcon Jet. Gather bandwidth per processing node as a function of the CM-5 configuration.

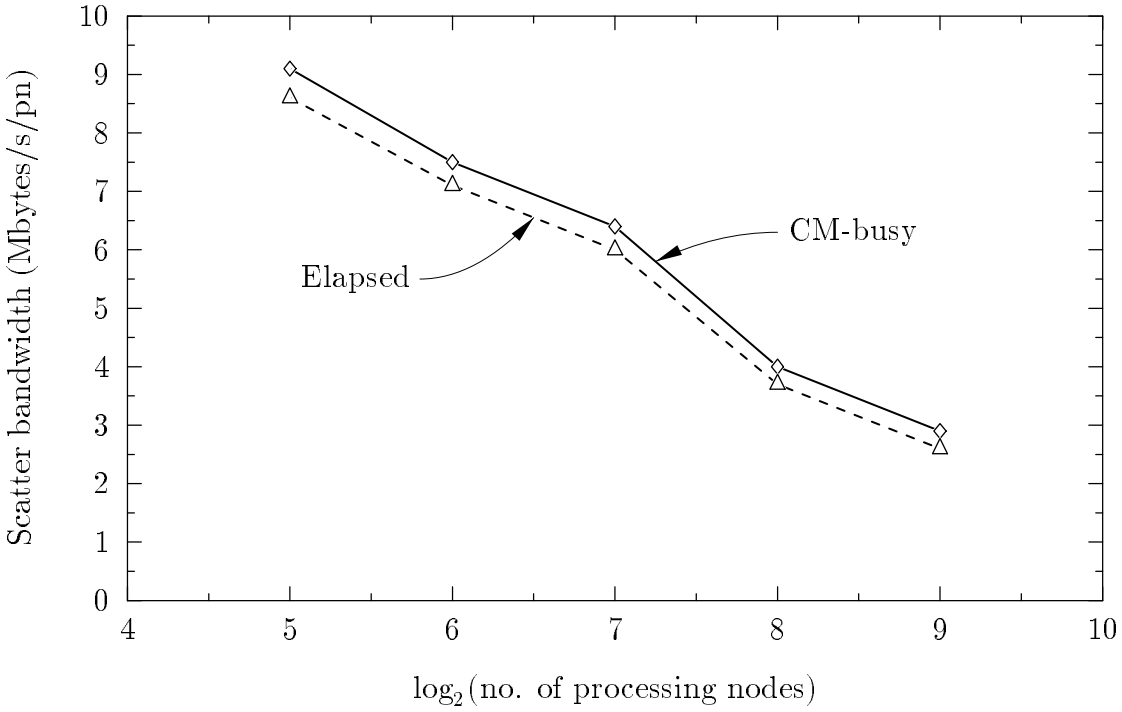


*Figure 13.* Falcon Jet. Scatter bandwidth per processing node as a function of the CM-5 configuration.

### 6.2. ONERA M6 wing

An ONERA M6 wing is placed in a Mach 0.84 inviscid flow at an angle of attack of 3.06 degrees. The mesh contains 48,011 nodes and 266,556 linear tetrahedra. Figure 14 presents a view of the surface mesh on the outer boundaries of the computational domain. One can see the high concentration of boundary elements on the plane of symmetry near the root of the wing.

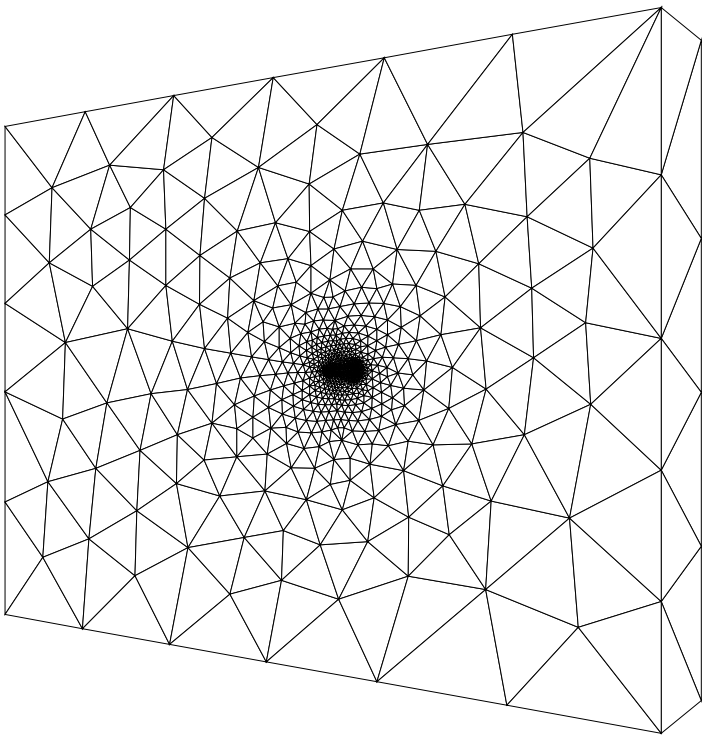

*Figure 14.* ONERA M6 wing. View of surface mesh on outer boundaries.

A uniform flow based on the free-stream conditions is used to initialize the finite element solver. The problem is run 20 time steps at a CFL number of 5 followed by 80 time steps at a CFL number of 10. The mesh and the pressure contours on the upper surface of the wing are shown in Figures 15 and 16. The classical flow pattern (two shocks merging near the wingtip) can be observed. The finite element program was run on a 64-node CM-5 system. A vectorized Fortran 77 version of this code was also run on 1 CPU of a Cray Y-MP C90 for performance comparison. Timings for both platforms are presented in Table 4. A few remarks can be made about these timings:

1. Partitioning the mesh takes 14% of the total time, making it possible to use for slowly

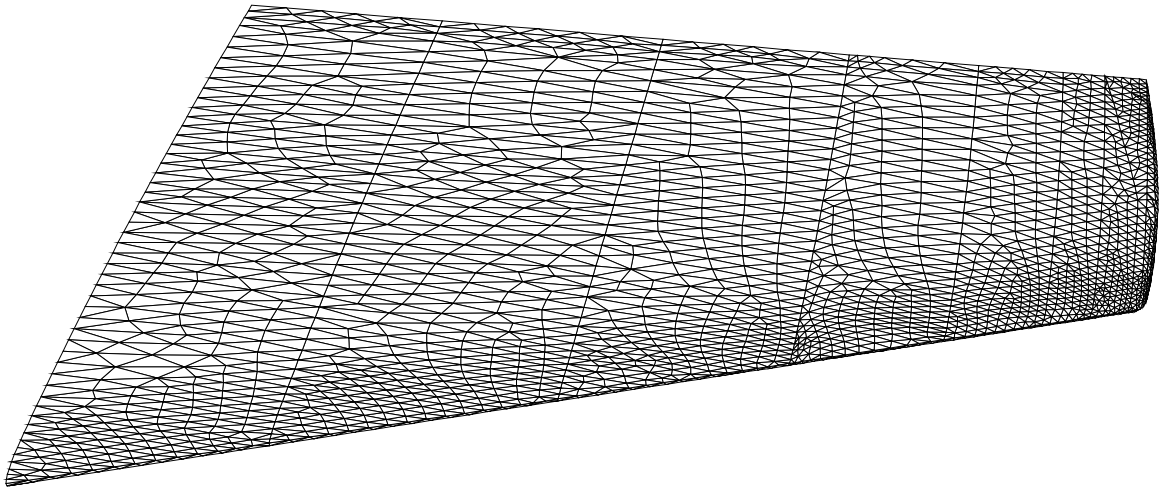

*Figure 15.* ONERA M6 wing. Mesh on wing upper surface.

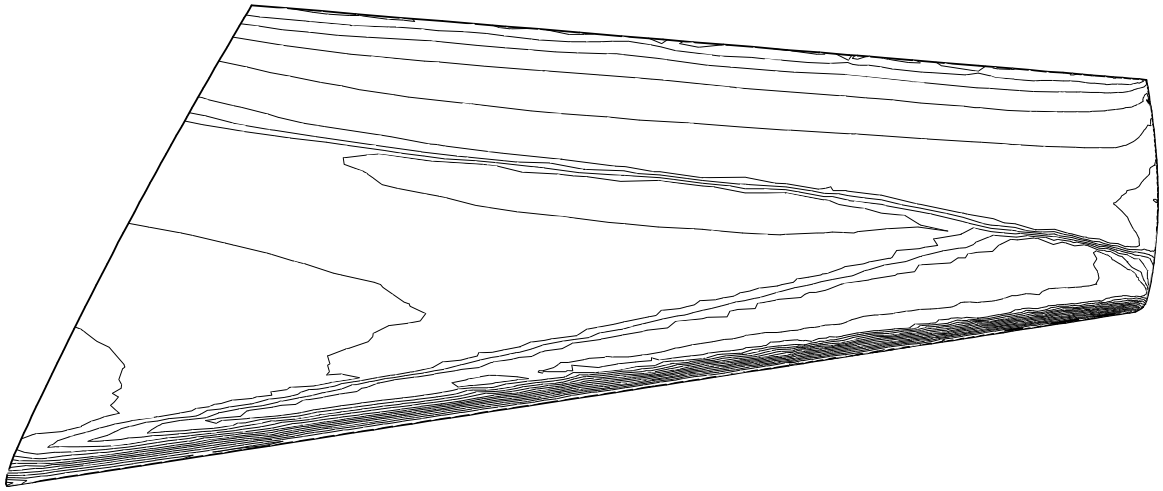

*Figure 16.* ONERA M6 wing. Pressure contours on wing upper surface.

adapting meshes. One example of such a case is mesh adaptation during the course of the computation to capture more accurately discontinuities in the flow-field.

2. The communication primitives represent 28% of the solver time (i.e., partitioning not included). The gather and scatter bandwidths ***per processing node*** are 12.3 Mbytes/s and 9.5 Mbytes/s, respectively. Note that these bandwidth numbers are weighted averages between memory bandwidth (used during local gather/scatter operations) and network bandwidth (used during global gather/scatter operations).
3. The performance per processing node of the computation part of the solver is 32.7 M-flops/s, which is 25% of the peak hardware performance. Upcoming enhancements of the CM Fortran compiler promise some improvement of the computational efficiency.
4. The overall efficiency of the solver for this example is 18% on the 64-node CM-5 system. This demonstrates that, contrary to the general belief, a reasonable efficiency can be achieved for finite element techniques on distributed-memory parallel systems.

*Table 4.* ONERA M6 wing. CM-busy times or CPU times for different parts of the finite element program run on a 64-node CM-5 system and a 1-CPU Cray Y-MP C90.

| | CM-5 | Cray Y-MP C90 |
|---|---|---|
| mesh partitioning | 123 s | — |
| gather operation | 77 s | — |
| computation | 528 s | 2532 s |
| scatter operation | 127 s | — |
| Total time | 14 min 15 s | 42 min 12 s |
| Solver flop rate | 1.5 Gflops/s | 0.44 Gflops/s |

### 6.3. Commercial aircraft

The inviscid flow past a two-engine commercial aircraft in cruise configuration (Mach 0.768, 1.116-degree angle of attack) is computed on a 128-node CM-5 system. Only half of the flow-field is computed because of symmetry properties. The mesh has 106,064 nodes and 575,986 linear tetrahedra. A view of the surface mesh on the half-airplane is shown in Figure 17. The problem is initialized by a uniform flow and converged for 100 time steps at a CFL number of 5. The Mach number contours on the airplane surface are shown in

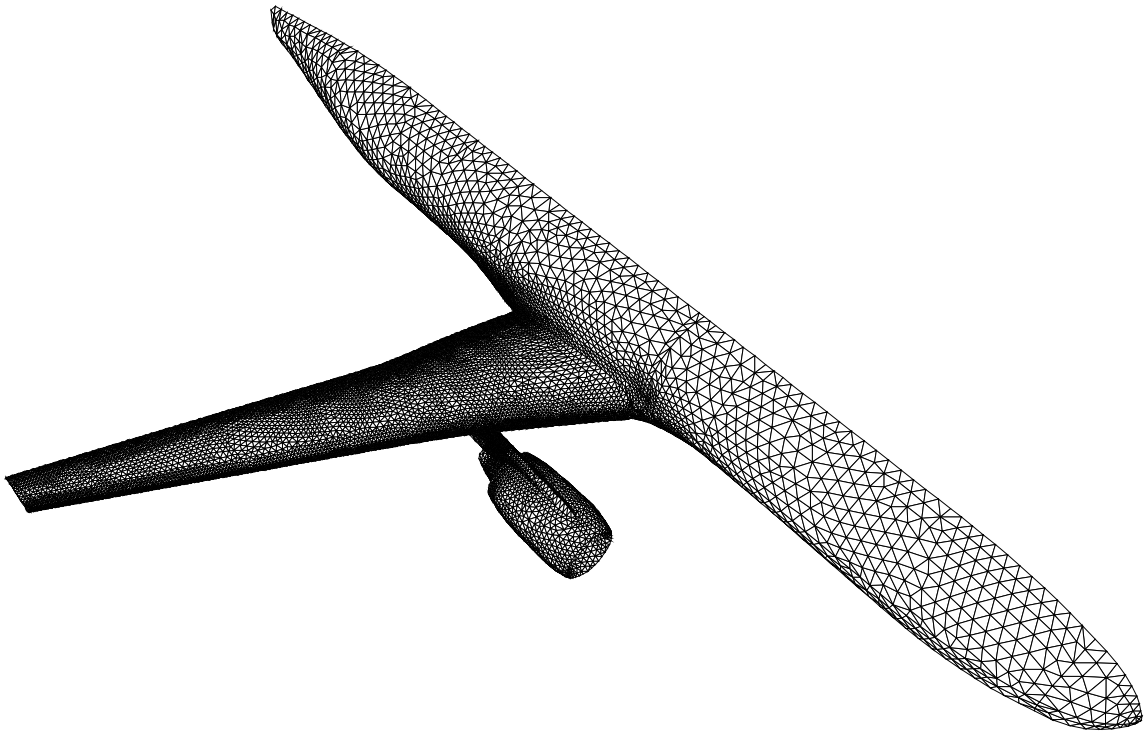

*Figure 17.* Commercial aircraft. View of surface mesh on the half-airplane.

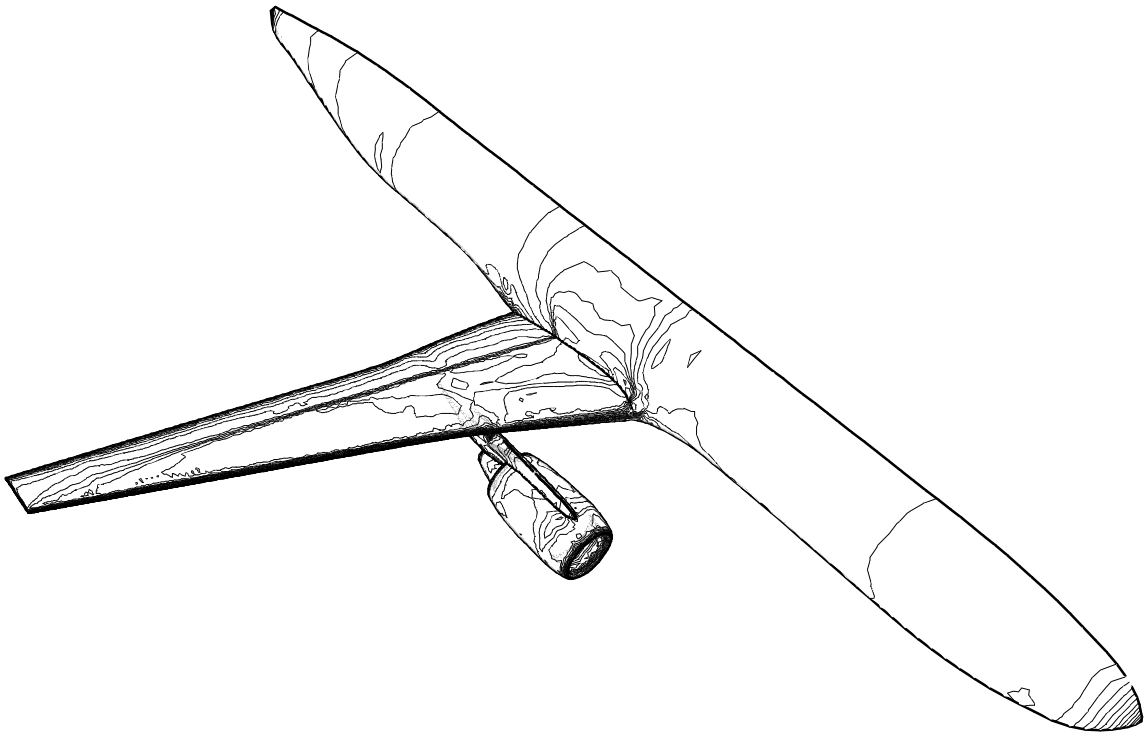

*Figure 18.* Commercial aircraft. Mach number contours on the airplane surface.

Figure 18. The solution is very close to the one obtained by Das *et al.* [3]. Timings for the mesh partitioning and the major parts of the solver are presented in Table 5. Remarks similar to the ONERA M6 wing case (see Section 6.2) can be made, namely a reasonable partitioning time and communication operations representing 26% of the solver time. The gather and scatter bandwidths per processing node are 13.8 Mbytes/s and 10.5 Mbytes/s, respectively. Finally, the aggregate performance of the solver is 3.1 Gflops/s.

*Table 5.* Commercial aircraft. CM-busy times for different parts of the finite element program run 100 time steps on a 128-node CM-5 system.

| | 128-node CM-5 |
|---|---|
| mesh partitioning | 196 s |
| gather operation | 58 s |
| computation | 454 s |
| scatter operation | 103 s |
| Total time | 13 min 31 s |

### 6.4. F-18 fighter jet

This last example solves the supersonic inviscid flow at Mach 1.5 around an F-18 fighter jet. The tetrahedral mesh has 182,055 nodes and 1,010,174 elements. The computation ran for 20 time steps at a CFL number of 5 followed by 80 time steps at a CFL number of 10. The resulting pressure contours on the surface of the airplane are shown in Figure 19. Note that we computed the fluid flow around the complete airplane even though this problem has a plane of symmetry. This problem was solved on a 256-node CM-5 system using both the run-time library available in CMOST 7.2 Beta 1.1 and the latest library to be included in CMOST 7.2 Beta 2 (referred to as 7.2 Beta 1.1 Patched in the table below). Timings for both software versions are presented in Table 6. It can be noted that this software upgrade yielded important improvements in partitioning and communication times, and additional speed-ups are to be expected before the final release of the CM-5 software. This numerical example also indicates that solving a 1 million degree of freedom aerodynamic problem under 20 minutes can be done routinely today on a medium-size parallel computer. Using the latest run-time library, the solver part of the finite element program was clocked at 6.3 Gflops/s on this CM-5 configuration. Gather and scatter bandwidths per processing node of 15.8 Mbytes/s and 10.1 Mbytes/s, respectively, have been obtained.

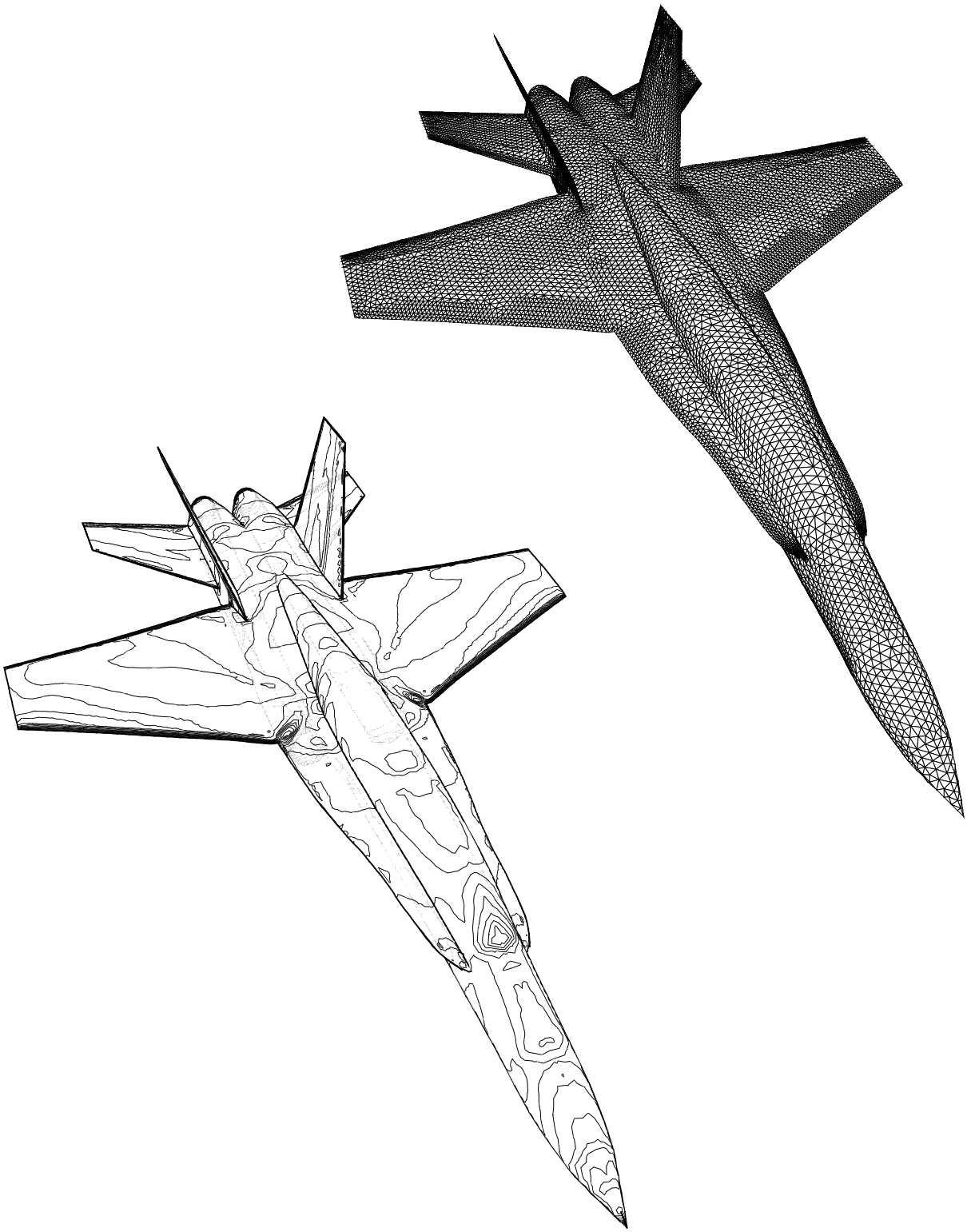

*Figure 19.* F-18 fighter jet. View of surface mesh and pressure contours.

*Table 6.* F-18 fighter jet. CM-busy times for different parts of the finite element program run 100 time steps on a 256-node CM-5 system.

| | 7.2 Beta 1.1 | 7.2 Beta 1.1 Patched |
|---|---|---|
| mesh partitioning | 345 s | 269 s |
| gather operation | 122 s | 65 s |
| computation | 565 s | 557 s |
| scatter operation | 154 s | 125 s |
| Total time | 19 min 46 s | 16 min 56 s |
| Solver flop rate | 5.6 Gflops/s | 6.3 Gflops/s |

### 6.5. Summary of performance results

Since the subgrid sizes (i.e., the number of elements per processing node) of the numerical examples presented above are all in the same range (3400-4500), a scalable architecture would render a linear speed-up as we go from one problem to another. This is indeed what we observe in Figure 20, which verifies the scalability of both the finite element program and the CM-5 system.

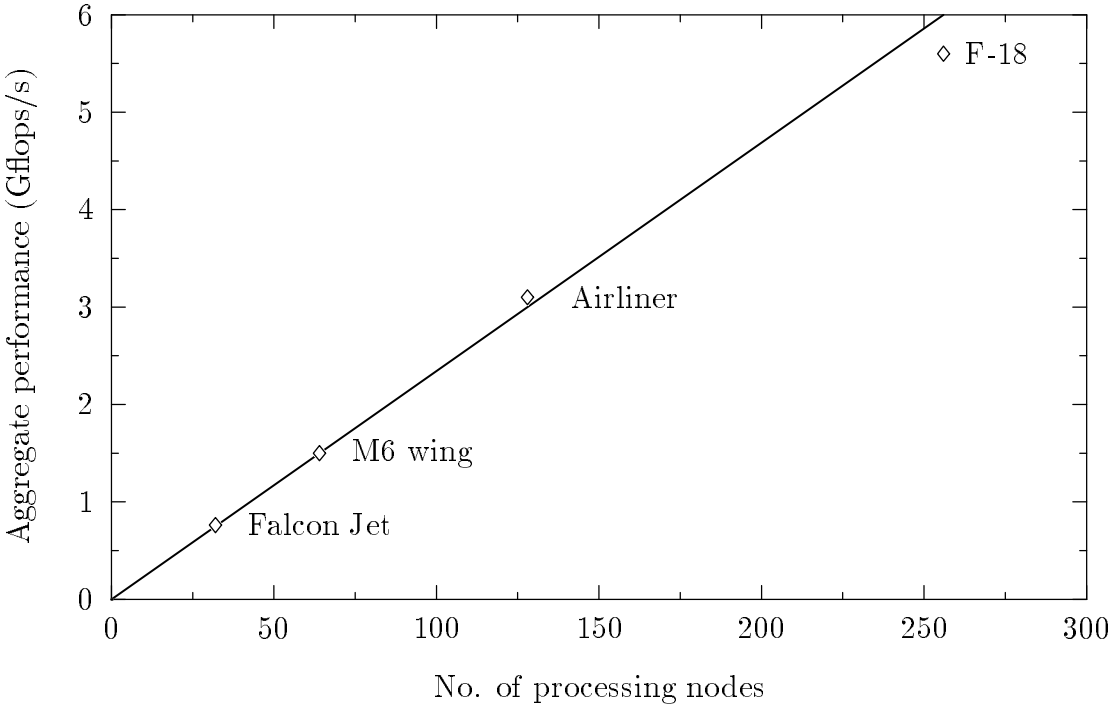


*Figure 20.* Summary of performance results as a function of the CM-5 configuration.

## 7. Conclusions

We have presented a new communication procedure for finite element methods on the CM-5 system. Proper and fast mapping of the data to the vector units of the CM-5 system, and taking advantage of that mapping in the design of the communication routines allowed us to achieve high performance on large-scale computational fluid dynamics applications. The mesh partitioner and the set of communication primitives have been made available to users in release 3.1 of the Connection Machine Scientific Software Library. One should note that, given these routines and a high-level language of the class of Fortran 90, finite element programs can be made architecture-independent. We have already achieved this independency between Connection Machine systems since porting our code to the CM-200 system can be done just through recompilation. Another important feature noted in this paper is that finite element programs using the communication primitives described herein will retain good scalability properties. This is a fundamental issue when developing software for massively parallel systems.

The next step in the design of universal communication tools for data-parallel finite element programs is related to dynamically changing meshes. Such changes happen either through rezoning (the mesh nodes move but the connectivity remains fixed) or remeshing (the connectivity changes), or both. In any case, improved partitioning strategies, parallel mesh generators and fast solution projection algorithms are required, and should be the focus of future work.

## Acknowledgements

We would like to thank Arthur Raefsky (CENTRIC Engineering Systems) for suggesting us to try the RSB algorithm, Horst Simon (NASA Ames) for giving us his sequential implementation of the RSB algorithm, and John Kennedy, Jacek Myczkowski and Richard Shapiro (Thinking Machines) for their helpful comments. We would like to express our gratitude to Rajiv Thareja (NASA Langley) for providing us with the cylinder mesh, Dassault Aviation for providing us with the Falcon Jet mesh, Jean Cabello and Rainald Löhner (George Washington University) for providing us with the ONERA M6 wing and F-18 meshes, and Dimitri Mavriplis (ICASE/NASA Langley) for providing us with the commercial aircraft mesh. We would also like to thank the Pittsburgh Supercomputing Center for providing us with computing time on a Cray Y-MP C90 and Yasunari Tosa (Thinking Machines) for running the F77 version of the code on this system. Access to

large CM-5 system configurations was provided by the National Center for Supercomputing Applications at the University of Illinois Urbana-Champaign.

## References

[1] S.T. Barnard and H.D. Simon, "A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems," Report RNR-92-033, NASA Ames Research Center, Moffett Field, CA, 1992.

[2] A.J. Beaudoin, P.R. Dawson, K.K. Mathur, U.F. Kocks and D.A. Korzekwa, "Application of polycrystal plasticity to sheet forming," submitted to *Computer Methods in Applied Mechanics and Engineering*.

[3] R. Das, D.J. Mavriplis, J. Saltz, S. Gupta and R. Ponnusamy, "The design and implementation of a parallel unstructured Euler solver using software primitives," *AIAA 30th Aerospace Sciences Meeting*, AIAA-92-0562, 1992.

[4] C. Farhat, N. Sobh and K.C. Park, "Dynamic finite element simulations on the Connection Machine," *International Journal of High Speed Computing*, **1** (1989) 289–302.

[5] M. Fiedler, "Algebraic connectivity of graphs," *Czechoslovak Mathematical Journal*, **23** (1973) 298–305.

[6] M. Fiedler, "Eigenvectors of acyclic matrices," *Czechoslovak Mathematical Journal*, **25** (1975) 607–618.

[7] M. Fiedler, "A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory," *Czechoslovak Mathematical Journal*, **25** (1975) 619–633.

[8] G.H. Golub and C.F. Van Loan, *Matrix computations*, Second edition, The Johns Hopkins University Press, Baltimore, MD, 1989.

[9] B. Hendrickson and R. Leland, "An improved spectral graph partitioning algorithm for mapping parallel computations," Report SAND 92-1460, Sandia National Laboratories, Albuquerque, NM, 1992.

[10] T.J.R. Hughes, L.P. Franca and G.M. Hulbert, "A new finite element formulation for computational fluid dynamics: VIII. The Galerkin/least-squares method for advective-diffusive equations," *Computer Methods in Applied Mechanics and Engineering*, **73** (1989) 173–189.

[11] T.J.R. Hughes, L.P. Franca and M. Mallet, "A new finite element formulation for computational fluid dynamics: I. Symmetric forms of the compressible Euler and

Navier-Stokes equations and the second law of thermodynamics," *Computer Methods in Applied Mechanics and Engineering*, **54** (1986) 223–234.

[12] T.J.R. Hughes, L.P. Franca and M. Mallet, "A new finite element formulation for computational fluid dynamics: IV. Convergence analysis of the generalized SUPG formulation for linear time-dependent multidimensional advective-diffusive systems," *Computer Methods in Applied Mechanics and Engineering*, **63** (1987) 97–112.

[13] T.J.R. Hughes and M. Mallet, "A new finite element formulation for computational fluid dynamics: III. The generalized streamline operator for multidimensional advection-diffusion systems," *Computer Methods in Applied Mechanics and Engineering*, **58** (1986) 305–328.

[14] Z. Johan, "Data parallel finite element techniques for large-scale computational fluid dynamics," *Ph.D. Thesis*, Stanford University, 1992.

[15] Z. Johan, T.J.R. Hughes, K.K. Mathur and S.L. Johnsson, "A data parallel finite element method for computational fluid dynamics on the Connection Machine system," *Computer Methods in Applied Mechanics and Engineering*, **99** (1992) 113–134.

[16] Z. Johan, T.J.R. Hughes and F. Shakib, "A globally convergent matrix-free algorithm for implicit time-marching schemes arising in finite element analysis in fluids," *Computer Methods in Applied Mechanics and Engineering*, **87** (1991) 281–304.

[17] C.E. Leiserson, Z.S. Abuhamdeh, D.C. Douglas, C.R. Feynman, M.N. Ganmukhi, J.V. Hill, W.D. Hillis, B.C. Kuszmaul, M.A. St. Pierre, D.S. Wells, M.C. Wong, S.-W. Yang and R. Zak, "The network architecture of the Connection Machine CM-5," *Proceedings of Parallel Algorithms and Architectures Symposium*, 1992.

[18] K.K. Mathur, "On the use of randomized address maps in unstructured three-dimensional finite element simulations," *Technical Report*, TMC-37/CS90-4, Thinking Machines Corporation, Cambridge, MA, 1990.

[19] K.K. Mathur and S.L. Johnsson, "Communication primitives for unstructured finite element simulations on data parallel architectures," *Computing Systems in Engineering*, **3** (1992) 63–71.

[20] B.N. Parlett, H. Simon and L.M. Stringer, "On estimating the largest eigenvalue with the Lanczos algorithm," *Mathematics of Computation*, **38** (1982) 153–165.

[21] A. Pothen, H.D. Simon and K.-P. Liou, "Partitioning sparse matrices with eigenvectors of graphs," *SIAM Journal of Matrix Analysis and Applications*, **11** (1990) 430–452.

[22] Y. Saad and M.H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal of Scientific and Statistical Computing*, **7** (1986) 856–869.

[23] F. Shakib, T.J.R. Hughes and Z. Johan, "A multi-element group preconditioned GMRES algorithm for nonsymmetric systems arising in finite element analysis," *Computer Methods in Applied Mechanics and Engineering*, **75** (1989) 415–456.

[24] H.D. Simon, "Partitioning of unstructured problems for parallel processing," *Computing Systems in Engineering*, **2** (1991) 135–148.

[25] A. Sussman, J. Saltz, R. Das, S. Gupta, D. Mavriplis, R. Ponnusamy and K. Crowley, "PARTI primitives for unstructured and block structured problems," *Computing Systems in Engineering*, **3** (1992) 73–86.

[26] R.R. Thareja, K. Morgan, J. Peraire and J. Peiro, "A three-dimensional upwind finite element point implicit unstructured grid Euler solver," *AIAA 27th Aerospace Sciences Meeting*, paper AIAA-89-0658, 1989.

[27] V. Venkatakrishnan, H.D. Simon and T.J. Barth, "A MIMD implementation of a parallel Euler solver for unstructured grids," *The Journal of Supercomputing*, **6** (1992) 117–127.

[28] *The Connection Machine CM-5 technical summary*, Thinking Machines Corporation, Cambridge, MA, 1992.

[29] *CDPEAC: Using GCC to program in DPEAC*, Revision 1.1, Thinking Machines Corporation, Cambridge, MA, 1992.

[30] *CM Fortran reference manual*, Version 2.0 Beta, Thinking Machines Corporation, Cambridge, MA, 1992.

[31] *CM Fortran Utility Library reference manual*, Version 2.0 Beta, Thinking Machines Corporation, Cambridge, MA, 1993.

[32] *CMSSL for CM Fortran: CM-5 Edition*, Version 3.1 Beta 2, Thinking Machines Corporation, Cambridge, MA, 1993.